Service Oriented Computing and Applications manuscript No.
(will be inserted by the editor)

Log-based Mining Techniques Applied to Web Service Composition Reengineering ⋆

Walid Gaaloul[1], **Karim Baïna**[2], **Claude Godart**[3]

[1] DERI-NUIG
IDA Business Park, Galway, Ireland
e-mail: walid.gaaloul@deri.org
[2] ENSIAS, Université Mohammed V - Souissi
BP 713 Agdal - Rabat, Morocco
e-mail: baina@ensias.ma
[3] LORIA-INRIA-UMR 7503
BP 239, F-54506 Vandœuvre-les-Nancy Cedex, France
e-mail: godart@loria.fr

Received: date / Revised version: date

Abstract There is no doubt that SOA and BPM will continue to evolve dependently for the next ten years. Preparing common research infrastructures will require most important efforts of web service community researchers. Web services composition is becoming more and more chaotic, involving numerous interacting ad-hoc services through huge business processes. Analysing and Reengineering of complex Web Service Compositions will enable them to be well understood, controlled, and redesigned. Our contribution to this problem is a patterns mining algorithm which is based on a statistical technique to discover patterns from execution log. Our approach is characterised by a "local" patterns discovery that allows to cover partial results through a dynamic programming algorithm. Those local discovered patterns are then composed iteratively until discovering the Composite Web Service. The analysis of the disparities between the discovered model and the initial ad-hoc CS model (delta-analysis) enables initial design gaps to be detected and thus the Web service composition to be re-engineered.

Key words Composite Service Mining, Service Intelligence, Service Analysis, Service Validation, Service Reengineering Frameworks, Model Driven Reengineering, Workflow Patterns.

1 Introduction

Service Oriented Architecture, becoming indispensable in building and integrating growing and complex internet applications is suffering from its own success: lack of models, difficult maintainability, and poor monitoring tools. Our Web Service composition reengineering approach is

⋆ The work presented in this paper was partially supported by the EU funding under the SUPER project (FP6-026850) and by the Lion project supported by Science Foundation Ireland under Grant No. SFI/02/CE1/I131

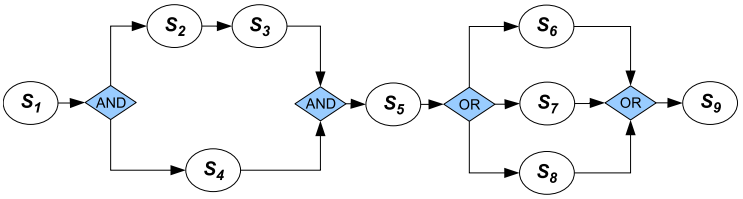
interested, among others, in two challenges: (i) post-execution discovering of complex web service orchestration patterns (e.g. identification of web service structural, and behavioural patterns), and (ii) runtime web service monitoring (e.g. web service QoWS -Quality of Web Service-, and KPI -Key Performance Indicators- analysis)[1]. Our composite web service reengineering approach aims to support *composite web service continuous evolution* by analysing composite web service execution logs, discovering effective web service compositions and helping to improve and to correct the initially designed web service based process models. Our paper is a contribution to this problem through our composite Service (CS) mining algorithm. Supporting business process rediscovery based on CS logs analysis, CS mining gathers retroactive (re)design models and techniques necessary to understand hidden business process execution reality.

The main idea of our algorithm is that a set of web services interact in an ad-hoc manner whose logics is distributed in their implementation code (i.e. with no explicit orchestration process). This ad-hoc web service interactions can be better abstracted and formalised as an orchestration process. Our purpose, through structural web service mining, is (1) to discover the implicit orchestration protocol behind a set of web services interactions, and (2) to explore to which extent this implicit protocol can be mapped to an explicit orchestration protocol (let say a BPEL process) either: (i) to be well managed and controlled, or (ii) to be well analysed and understood, or (iii) to be verified from either structural or behavioral point of view. Our algorithm starts by collecting log information from CS execution instances as they took place. Then, it builds, through statistical techniques, a graphical intermediary representation modelling service elementary dependencies. These dependencies are then refined to discover control flow patterns. The discovered results are used thereafter in reengineering and analysis phase.

Motivating example

In this article, we will illustrate our ideas using a running CS that was implemented, for example, as an ad-hoc composite web service orchestration. This CS represents a car rental application (see Figure 1). It acts as a broker offering to its customers a set of car choices made from their requirements expressed in the (web) service S_1. S_4 service checks the customer ID while S_2 service checks car availability and S_3 service provides available cars information and the respective car rental companies supplier. Afterwards, the customer makes his choice and agrees on rental terms in S_5 service. The customer is requested to pay either by credit card (S_6), by check (S_7), or by cash (S_8) or by combining the payment by credit card and by check. Finally, the bill is sent to the customer by S_9 service.

Figure 1 CS running example : an ad-hoc composite web service orchestration



[1] the second challenge is not in the scope of this paper.

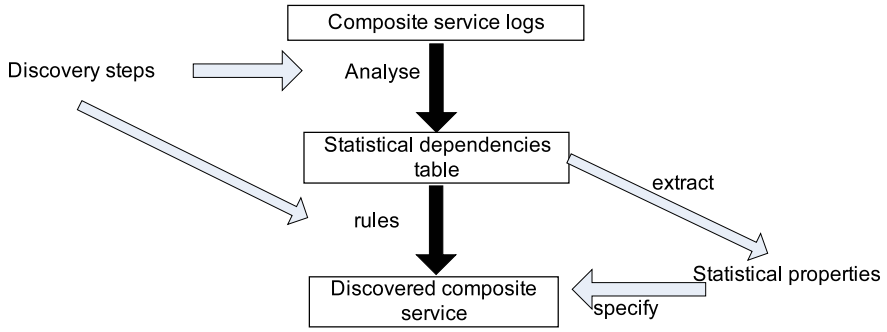
Generally, previous approaches develop, based on their modeling formalisms, a set of techniques to analyze and check the composition model. Although powerful, these approaches may fail, in some cases, to ensure CS reliable executions even if they formally validate the CS model. This is because properties specified in the studied composition models remains assumptions that may not coincide with the reality. In fact, the users during execution can express different needs from the initial ad-hoc CS model, by choosing a different payment process or removing a concurrent behavior. Formal approaches cannot report these dynamic requirements and needs or deal with such scalable design features.

Overview

In this article, we describe a set of mining techniques and algorithms which we have specified, proofed and implemented for CS patterns discovery. This approach allows to detect and correct design errors due omissions or errors at the initial (ad-hoc) design phase, or CS evolutions observed at the run time. With this intention, we rediscover from CS log the effective or "real" service interactions reflecting execution "reality". The following step presents an overview of our approach:

- **Collecting execution history:** The purpose of this phase is keeping track of the composite service execution by capturing the relevant generated events. The number of instances and the log structure should be sufficiently rich to enable CS mining.
- **Analyzing the execution history:** The purpose of this phase is mining the effective control flow of a composite service. Concretely, to discover the CS model based only on its log, we proceed in two steps (see Figure 2). The first consist in log statistical analyses to extract statistical dependencies tables. The second steps consists in a statistical specification of the control flow properties extracted using discovery rules to apply on the statistical dependencies tables.
- **Improving the composition model:** Based on the execution history analysis and the mined results we enable a reengineering phase to improve the composite service model.

Figure 2 Patterns mining steps



To describe our algorithm, the next section 2 explains our CS log model. Section 3 details our structural control flow patterns mining algorithm. Validation and reengineering elements of the initial designed process model with the discovered process model are given by section 4. Implementation aspects are discussed in section 5. Finally, section 6 discusses related work, and perspectives issues, before concluding.

2 Composite Service Log

In the following, we are interested on CS log related issues. We describe techniques to collect WS logs. Then, we present in section 2.2 our CS log model. Thereafter, we propose in section 2.3 the log structure and the minimal conditions that log should respect, to be able to apply our control flow mining algorithm.

2.1 Collecting Web service logs

Following a common requirement in the areas of business processes and services management, we expect the composite services to be traceable, meaning that the system should in one way or another keep track of ongoing and past executions. Several research projects deal with the technical solutions necessary for the collecting and the logging of Web services execution log [1–3]. In the following, we examine and formalize the logging possibilities in service oriented architectures which is a requirement to enable the approach described in this paper.

2.1.1 Web service collecting solutions The first step in the Web Service mining process consists of gathering the relevant Web data, which will be analyzed to provide useful information about the Web Service behavior. We discuss how these log records could be obtained by using existing tools or specifying additional solutions. Then, we show that the mining abilities is tightly related to what of information provided in web service log and depend strongly on its richness.

Traditional Web logging solutions provide a set of tools to capture web services logs. These solutions remain quite "poor" to mine advanced web service dependencies. That is why advanced Web Service logging solutions should propose a set of developed techniques that allows us to record the needed information to mine more advanced behavior. This additional information is needed in order to be able to collect and distinguish particulary between CS execution instances.

2.1.2 Traditional Web service logging solutions There are two main sources of data for Web log collecting, corresponding to the two interacting software systems: the Web server side and the client side. The existing techniques are commonly achieved by enabling the respective Web server's logging facilities. There are many investigations and proposals on Web server log and associated analysis techniques. Actually, papers on Web Usage Mining [4] describe the most well-known means of web log collection. Basically, server logs are either stored in the *Common Log Format* [2] or the more recent *Combined Log Format* [3]. They consist primarily of various types of logs generated by the Web server. Most of the Web servers support as a default option the *Common Log Format*, which is a fairly basic form of Web server logging.

However, the emerging paradigm of Web services requires richer information in order to fully capture business interactions and customer electronic behavior in this new Web environment. Since the Web server log is derived from requests resulting from users accessing pages, it is not tailored to capture service composition or orchestration. That is why, we describe in the following a set of advanced logging techniques that allows to record the additional information to mine more advanced behavior.

[2] http://httpd.apache.org/docs/logs.html
[3] http://www.w3.org/TR/WD-logfile.html

2.1.3 Advanced Web service logging solutions CS mining requires choreography or orchestration identifier and instance (case) identifier in the log record. Such information is not available in conventional Web server logs. In the following, we describe advanced solution to collect this information in choreography or orchestration execution.

A known method for debugging, is to insert logging statements into the source code of each service in order to call another service or component, responsible for logging. However, this solution has a main disadvantage: we do not have ownership over third parties code and we cannot guarantee they are willing to change it on someone else behalf. Furthermore, modifying existing applications may be time consuming and error prone (solution 1). Since all interactions between Web Services happen through the exchange of SOAP message, an other alternative is to use SOAP headers that can provide additional information on the message's content concerning the executed choreography. Basically, we modify SOAP headers to include and gather the additional needed information capturing choreography-ID and its instance-ID. We use SOAP intermediaries [5] which are an application located between a client and a service provider. These intermediaries are capable of both receiving and forwarding SOAP messages. They are located on web services provider and they intercept SOAP request messages from a Web service sender or capture SOAP response messages from a Web service provider. On Web service client-side, this remote agent can be implemented to intercept those messages and extract the needed information. The implementation of client-side data collection methods requires user cooperation, either in enabling the functionality of the remote agent, or to voluntarily use and process the modified SOAP headers but without changing the Web service implementation itself (the disadvantage of solution 1).

For **orchestration** log collecting, since the most web services orchestration are using a WS-BPEL engine, which coordinates the various orchestration's web services, interprets and executes the grammar describing the control logic, we can extend this engine with a sniffer that captures orchestration information, i.e., the orchestration-ID and its instance-ID. This solution is centralized, but less constrained than the previous one which collects choreography information.

Using these advanced logging facilities, we aim at taking into account web services' neighbors in the mining process. The term neighbors refers to other Web services that the examined Web Service interacts with. The concerned levels deal with mining web service choreography interface (abstract process) through which it communicates with others web services to accomplish a choreography, or discovering the set of interactions exchanged within the context of a given choreography or composition.

The exact structure of the web logs or the event collector depends on the used execution engine. In our experiments, we have used the bpws4j[4] engine which uses log4j[5] to generate logging events. Log4j is an OpenSource logging API developed under the Jakarta Apache project. It provides a robust, reliable, fully configurable, easily extendible, and easy to implement framework for logging Java applications for debugging and monitoring purposes. The event collector (which is implemented as a remote log4j server) sets some log4j properties of the bpws4j engine to specify level of event reporting (INFO, DEBUG etc.), and the destination details of the logged events. At runtime bpws4j generates events according to the log4j properties set by the event collector.

2.2 Composite Service log structure

Definition 1 defines our CS log model converted from the event collector described in the previous section to select only the required information. A CSLog is composed of a set of EventStreams.

[4] http://alphaworks.ibm.com/tech/bpws4j
[5] http://logging.apache.org/log4j

Each EventStream traces the execution of one case (instance). It consists of a set of events (Event) that captures services execution. WS logging functionalities might collect external events that capture the service life cycle (such as activated, aborted, failed, and terminated). However, existing logging solutions can propose different level of granularity and collects only one part of the set of event states. Moreover, the nomenclatures of these states are generally different from one system to another. As solution, we can simply choose to filter them or/and designate them as a default state. In our case (i.e. we aim to only mine the control flow), it is more practical to simply consider ordered service atomic terminated state events, which omit execution times and other intermediate service states (simplified EventStream).

Although the combination of activated and terminated states can be very useful to detect services concurrent behavior implicitly from logs, our log structure report only the event of successful termination, to simplify and to minimize the constraints of log collecting. This "minimalist" feature enables us to exploit "poor" logs which contain only information concerning the services sequence executed successfully without collecting for example services execution intermediate states or execution occurrence times.

Definition 1 (CSLog)
An event reports a service terminated *state and its related execution time, and is defined as a tuple:* Event= *(*serviceId, TimeStamp*). An* EventStream *represents the history of a CS instance events as a tuple* EventStream= *(*sequenceLog, SOccurrence*) where:*
✓sequenceLog: Event**; is an ordered* Event *set reporting service executions;*
✓SOccurrence: int*; is the instance ID.*
A CSLog *is a set of* EventStream*s.* CSLog=*(*ID, {EventStream$_i$, $0 \leq i <$ *number of CS instances*}*) where* EventStream$_i$ *is the event stream of the* i^{th} *CS instance.*
Let $\mathcal{T}$ *the set of services belonging to a CS. We note* $\sigma \in \mathcal{T}^*$ *a simplified* EventStream *format by omitting* TimeStamp *from* Event *as* Event*s are ordered according to their occurrence time.*

2.3 Sufficient and minimal number of CS instances

To enable correctly the mining process, CS logs must be "complete" by respecting the *log completeness conditions* [6]. Basically these condition express that if a service precedes another in the control flow then there should be one instance log, at least, reporting two respective related events keeping the same order. In particular, if the execution of one service depends directly on the termination of another service, then the event related to the first service must *directly* (immediately, without intercalated event between them) follow at least once the event related to the second service in an instance log. To discover the parallel behavior of two concurrent services with a lack of indication related to the services begin and end execution time, we require that the events of the two parallel services should appear at least in two EventStreamss without order of precedence. Basically, two parallel services must *directly* follow each other in different order in two instances.

Based on this, we have specified "complete" logs features describing the properties required by our control flow mining approach from logs. Concretely, we have specified "minimalist" conditions on log structure and "sufficient" conditions on log quantity (i.e., number of logged instances) to be "complete". We deducted, for a given CS, the sufficient number of different instances logs for a "complete" CS log (*c.f.* lemma 1). This lemma indicates for each behavior, the necessary number

of instances logs to enable our control flow mining approach. This faculty defines 'a 'local specification" on the sufficient number of instances logs for a "complete" CS log. Thanks to this, we do not require to collect all possible instances logs to satisfy the *log completeness conditions*. For example, for a CS containing n concurrent services followed by m concurrent services, the number of possible scenarios (number of instances logs) is equal to $n! * m!$. But, by applying our lemma we need only $n! + m!$ instances logs.

Lemma 1 (Number of instances for a complete log) *The sufficient number of different* EventStream*s to discover a CS's control flow is computed as follows:*

1. *The minimal number is equal to 1. For example, a CS containing only one sequential services flow without concurrent or conditional behavior reports always the same sequence of services;*
2. *A conditional behavior between n services before a "*join*" point or after a "*fork*" point of requires $n-1$ different additional* EventStream*s.*
3. *A concurrent behavior between n control flows, each flow i; $0 < i < n+1$ contains na_i; $na_i \leq na_{i+1}$ services, requires $(\Pi_{i=1..n}(na_i+i-1))-1 = na_1*(na_2+1)*(na_3+2)*(na_4+3)*.....*(na_n+n-1)-1$ different additional* EventStream*s.*

Table 2 represents the execution of six instances of our running example. This table contains the sufficient information which we suppose to be present to be able to apply correctly our control flow mining approach. Indeed, our CS example contains a conditional behavior between three services (S_6, S_7, and S_8), which implies 3 EventStreams 1+(3-1)=3 by applying the second point of our lemma and by adding the necessary instance to satisfy the first point. In addition, the two concurrent flows, containing respectively S_2 et S_3, and S_4 imply (1*(2+1))-1= 2 additional EventStreams by applying the third point of our lemma. Finally, the parallel behavior that can be observed between S_6 and S_8 if the user decides to pay using credit card and cash implies (1*(1+1))-1= 1 additional EventStreams. The total of sufficient EventStreams will be equal to 3+2+1 = 6, which Table 1 reports.

In this table, the EventStreams 1 and 4 describe the case where the user chose to pay by check. Although these different EventStreams (1 and 4) describe the same scenario, the concurrent services execution scenario is not the same one (i.e services S_2, S_3 and S_3 do not have the same order termination each time). These different EventStreams, (in the same way for the EventStreams 5 and 6) allow to describe the various possible choices of the processing as well as the various possible combinations of concurrent services execution in these choices.

Table 1 6 simplified EventStreams of our motivating example

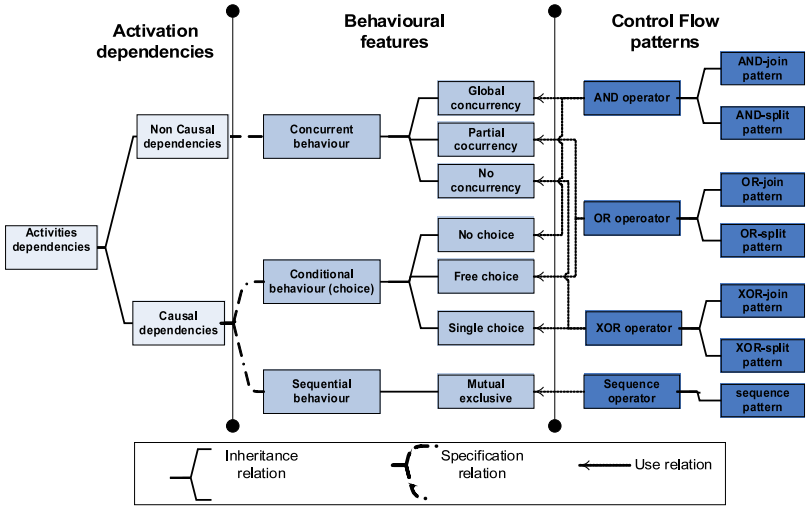
Instance ID	EventStream
Instance1	$S_1\ S_2\ S_3\ S_4\ S_5\ S_7\ S_9$
Instance2	$S_1\ S_2\ S_3\ S_4\ S_5\ S_6\ S_9$
Instance3	$S_1\ S_2\ S_3\ S_4\ S_5\ S_8\ S_9$
Instance4	$S_1\ S_2\ S_4\ S_3\ S_5\ S_7\ S_9$
Instance5	$S_1\ S_4\ S_2\ S_3\ S_5\ S_6\ S_7\ S_9$
Instance6	$S_1\ S_4\ S_2\ S_3\ S_5\ S_7\ S_6\ S_9$

3 Mining structural control flow patterns

The control flow (or skeleton) of a CS specifies the partial ordering of component services activations. We use (workflow-like) patterns to define a composite service skeleton. As defined in [7], a pattern is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts". A workflow pattern [8] can be seen as an abstract description of a recurrent class of interactions based on (primitive) activation dependency. In the following, we describe a bottom up approach, as illustrated in Figure 3, to discover these patterns:

1. **Discovering activation dependencies:** First, we specify dependencies linking the component services during execution. These dependencies are of two kinds: causal and non-causal. A Causal dependency between two services expresses that the occurrence of a service event involves the activation of an other service event. While a non-causal dependency specifies any other services behavioral dependency expressed a non causal dependency (concurrent behavior, for instance).
2. **Computing statistical behavioral properties:** Secondly, we compute the statistical behavioral properties from log. These properties tailor the main behavior features of the discovered patterns. These properties are of three types: "sequential", "concurrent" and "choice". The "sequential" and "choice" properties inherit from causal dependency. The first expresses an exclusive causal dependency between two services. While the second specifies a causal dependency between a service on one hand and one or many services belonging to a set of services, on an other hand. The "concurrent" property inherits from non-causal dependency and characterises the concurrent behavior of a set of services.
3. **Discovering control flow patterns:** Finally, we use a set of rules to discover a set of the most useful patterns. These rules are expressed using the statistical properties and could be expressed as a 1^{st} order logic predicate, for instance. In this work, we have chosen to discover the most useful patterns: sequence, xor-split, and-split, or-split, xor-join , and-join and M-out-of-N-Join. But the adopted approach allows to enrich this set of patterns by specifying new statistical dependencies and their associated properties or by using the existing properties in new combinations.

Figure 3 Hierarchical view of our patterns mining approach



We note that the only input of our mining approach is CS log. In the following, we suppose, after sufficient execution cases, that the CS log collecting cannot report new different cases, and the collected log should be complete for the *discovered* CS. However, this log could not be complete for the initially designed ad-hoc CS model and than cannot faithfully mine this model (see section 4 for more details).

3.1 Discovering activation dependencies

The aim of this section is to explain our algorithm for discovering activation dependencies among a CSLog and build an intermediary model representing these dependencies: the statistical dependency table (SDT).

3.1.1 Discovering direct dependencies A direct dependency is an "immediate" dependency linking two services in the sense that the termination of the first causes *directly* the activation of the last. Thus, the event of termination of the first service is considered as the pre-condition of the activation of the last and reciprocally the activation of the last is considered as a post condition of the termination of the first service. In order to discover direct dependencies from a CSLog, we need an intermediary representation of this CSLog through a statistical analysis. We call this intermediary representation: statistical dependency table (SDT) which is based on a notion of frequency table [9].

Algorithm 1 Computing initial SDT

```
1: procedure COMPUTINGSDT(SDT, MSDT)
2: input: CSLog: CSLog
3: output: #: service occurrence table ; SDT: Statistical dependency table;
4: Variable: stream_size: int; TDS_size: int;
5: #: int[]; depFreq: int[][];                                   ▷ initialized to 0;
6:
7:     for every stream: EventStream in CSLog do
8:         stream_size ← stream.size();          ▷ size returns the number of services in a stream
9:         for int i=1; i < stream_size; i++; do
10:            #[stream.get(i)]++;                          ▷ get returns the service whose index is i
11:            depFreq[stream.get(i)][stream.get(i-1)]++;
12:        end for
13:    end for
14:    SDT_size = Size-tab(#); /*return the size of #*/
15:    for int j=0; j < SDT_size; j++; do
16:        for int k=0; k < SDT_size; k++; do
17:            SDT[j, k] ← depFreq[j][k]/ #[j];
18:        end for
19:    end for
20: end procedure
```

Basically, SDT is built through statistical calculus that extracts event direct dependencies. For each service S, we extract from CSLog the following information in the statistical dependency table (SDT): (i) The overall occurrence number of this service (denoted $\#S$) and (ii) The elementary dependencies to previous services S_i (denoted $P(S/S_i)$). The size of SDT is $n * n$, where n is the number of component services. The (*m,n*) table entry is the frequency of the n^{th} service immediately

preceding the m^{th} service. Based on this, Algorithm 1 computes the "initial" SDT (Table 2) of our motivating example given in Figure 1. For instance, in this table P(S_3/S_2)=0.69 expresses that we have 69% of chance that S_2 occurs directly before S_3 in the log. This table was computed using 100 EventStreams captured after executing 100 instances (cases) of our motivating example[6].

Table 2 Initial Statistical Dependencies Table ($P(x/y)$) and Service Frequencies ($\#$)

$P(x/y)$	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S_1	0	0	0	0	0	0	0	0	0
S_2	**0.54**	0	0	0.46	0	0	0	0	0
S_3	0	**0.69**	0	0.31	0	0	0	0	0
S_4	**0.46**	0.31	0.23	0	0	0	0	0	0
S_5	0	0	**0.77**	**0.23**	0	0	0	0	0
S_6	0	0	0	0	1	0	0	0	0
S_7	0	0	0	0	1	0	0	0	0
S_8	0	0	0	0	0	0	0	0	0
S_9	0	0	0	0	0	0.38	0.62	0	0

$\#S_1 = \#S_2 = \#S_3 = \#S_4 = \#S_5 = \#S_9 = 100,$
$\#S_6 = 38, \#S_7 = 62, \#S_8 = 0$

We demonstrated a correlation between the service activation dependencies and the log statistics expressed in SDT (see Theorem 1). Each dependency between two services is expressed by a positive value in the corresponding SDT entry. This expresses a relation of equivalence between the positive entries in SDT and the dependencies between the related services.

Theorem 1 (Correlation between SDT and services dependencies) *Let wft a CS whose control flow is composed using the set of 7 enounced patterns and does not contain short loops.* $\forall a, b \in wft$ *where a precedes* $b \Leftrightarrow P(b/a) > 0 \wedge P(a/b) = 0$

Proof (Proof of Theorem 1:)
Proofing first right implication "$\Rightarrow$"

Let L the CSLog capturing a and b execution. By applying the *log completeness conditions* specified in section 2.3 we can deduce that:

$$\exists \sigma_1 = t_1 t_2 t_3 .... t_{n-1} \in L \wedge \exists 0 < i < n, | t_i = a, t_{i+1} = b$$

And through SDT building definition and based on instance log, we can deduce that:

$$a \prec b \Rightarrow P(b/a) > 0$$

Furthermore, supposing now (proof by contradiction) that $P(a/b) > 0$ this implies:

$$\exists \sigma_1 = t_1 t_2 t_3 .... t_{n-1} \in L \wedge \exists 0 < i < n, | t_i = b, t_{i+1} = a$$

[6] Table 1 is not used to compute this table

and as we have yet $P(b/a) > 0$ this implies:

$$\exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, | t_i = a, t_{i+1} = b$$

However this is absurd because this case arises only in a short[7] loop or if we have concurrent services. Thus we have:

$$a \prec b \Rightarrow P(b/a) > 0 \wedge P(a/b) = 0$$

Proofing second left implication "$\Leftarrow$" (proof by contradiction)

We have $P(b/a) > 0 \wedge P(a/b) = 0$, thus based on the SDT building definition we can deduce:

$$\exists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, | t_i = a, t_{i+1} = b$$
$$\wedge \nexists \sigma_1 = t_1 t_2 t_3 ....t_{n-1} \in L \wedge \exists 0 < i < n, | t_i = a, t_{i+1} = b \textbf{ (1)}$$

And as a and b belong to the set of the 7 described patterns, two sub cases happen if they are causally independent:

1. The two services a et b belong to two different separated patterns. This is absurd based on instance log **(1)** that shows that the two services happen one after the other.
2. The two services a et b are in concurrence. This is absurd based on instance log **(1)** and the *log completeness conditions*.

Thus a precedes b. Indeed, the case b precedes a is trivially absurd by applying "$\Rightarrow$" way. In conclusion, we have:

$$a \prec b \Leftarrow P(b/a) > 0 \wedge P(a/b) = 0$$

But as it was calculated, SDT presents some problems to express "correctly" and "completely" services dependencies related to the *concurrent* and the *conditional* behavior. Indeed, these entries are not able to identify the *conditional* behavior and to report he *concurrent* behavior pertinently. In the following, we detail these problems and we propose solutions to correct and complete these statistics.

3.1.2 Discarding erroneous dependencies If we assume that each EventStream from CSLog comes from a sequential (i.e no concurrent behavior) CS, a zero entry in SDT represents a causal independence and symmetrically a non-zero entry means a causal dependency relation (*i.e.* sequential or conditional relation). But, in case of concurrent behavior, as we can see in patterns (like and-split, and-join, etc.), the EventStreams may contain interleaved events sequences from concurrent threads. As consequence, some entries in initial SDT can indicate non-zero entries that do not correspond to causal dependencies. For instance, the EventStream 4 in Table 1 "suggests" erroneous causal dependencies between S_2 and S_4 in one side, and between S_4 and S_3 in another side. Indeed, S_2 comes immediately before S_4 and S_4 comes immediately before S_3 in this EventStream. These erroneous entries are reported by $P(S_4/S_2)$ and $P(S_3/S_4)$ in initial SDT which are different to zero. These entries are erroneous because there are no causal dependencies between these services as suggested (i.e. noisy SDT). Underlined values in Table 2 report this behavior for other similar cases.

Formally, based on the *log completeness conditions*, we can easily deduce that form a complete CS log two services A and B are in concurrence *iff* $P(A/B)$ and $P(B/A)$ entries in SDT are non-zero entries in SDT. Based on this, we propose an algorithm to discover services parallelism and then mark the erroneous entries in SDT. Through this marking, we can eliminate the confusion caused by

[7] contain one or two services

the concurrent behavior producing these erroneous non-zero entries. The algorithm 2 scans the initial SDT and marks concurrent services dependencies by changing their values to (-1). For instance, we can deduce from Table 2 that S_2 and S_4 services are in concurrence (i.e $P(S_2/S_4) \neq 0 \wedge P(S_4/S_2) \neq 0$), so after applying our algorithm $P(S_2/S_4)$ and $P(S_4/S_2)$ will be equal to -1 in the final table. The algorithm's output is an intermediary table that we called marked SDT (MSDT).

Algorithm 2 Marking concurrent services in SDT

```
1: procedure MARKINGSDT(SDT, MSDT)
2: input: SDT Statistical dependencies table
3: output: MSDT Marked Statistical dependencies table
4: Variable: MSDT_size: int;
5:
6:     MSDT ← SDT;
7:     MSDT_size ← Size-tab(MSDT);                    ▷ calculates MSDT size
8:     for int i=0; i< MSDT_size; i++; do
9:         for int j=0; j<i; j++; do
10:            if SDT[i][j] > 0 ∧ SDT[j][i]>0 then
11:                MSDT[i][j] ← -1;
12:                MSDT[j][i] ← -1;
13:            end if
14:        end for
15:    end for
16: end procedure
```

3.1.3 Discovering indirect dependencies For concurrency reasons, a service might not depend on its immediate predecessor in the EventStream, but it might depend on another "indirectly" preceding services. As an example of this behavior, S_4 is logged between S_2 and S_3 in the EventStream 4 in the Table 1. As consequence, S_2 does not occur always immediately before S_3 in the CS log. Thus, we have only $P(S_3/S_2) = 0.69$ that is an under estimated dependency frequency. In fact, the right value is 1 because the execution of S_3 depends exclusively on S_2. Similarly, values in bold in initial SDT report this behavior for other cases.

Definition 2 (Concurrent window) *Formally, a concurrent window defines a triplet* ***window****(wLog, bWin, eWin) as a log slide over an* EventStream *S:* EventStream *(bStream, eStream, sLog, CSocc) where:*

- *bStream* $\leq$ *bWin* $\wedge$ *eWin* $\leq$ *eStream*
- *wLog* $\subset$ *sLog and* $\forall$ *e:* ***event*** $\in$ *S.sLog where bWin* $\leq$ *e.TimeStamp* $\leq$ *eWin* $\Rightarrow$ *e* $\in$ *wLog.*

We define the function width(***window***) *which returns the number of services in the* ***window****.*

To discover these indirect dependencies, we introduce the notion of *concurrent window* (Definition 2). An *concurrent window* (CW) is related to the service of its last event covering its whole preceding services. Initially, the width of CW of a service is equal to 2. Each time, this service is in concurrence with an other service we add 1 to this width. If this service is not in concurrence with other services and has preceding concurrent services, then we add their number to CW width. For example, S_4 is in concurrence with S_2 and S_3 the width of its CW is equal to 4. Based on this, the al-

gorithm 3 computes the *concurrent window* of each service grouped in the CW table. This algorithm scans the "marked" **SDT** calculated in last section and updates the CW table in consequence.

Algorithm 3 Calculating concurrent window size

```
1: procedure WINDOWWIDTH(MSDT, ACWT)
2: input: MSDT: Marked Statistical dependencies table
3: output: ACWT: CW size table
4: Variable: MSDT_size: int;
5:
6:     MSDT_size ← Size-tab(MSDT);                    ▷ calculates MSDT size
7:     for int i=0; i< MSDT_size; i++; do
8:         ACWT[i]=2;
9:     end for
10:    for int i=0; i< MSDT_size; i++; do
11:        for int j=0; j < MSDT_size; j++; do
12:            if MSDT[i][j] =-1 then
13:                ACWT[i]++;
14:                ACWT[j]++;
15:            end if
16:            for int k=0; k< MSDT_size; k++; do
17:                if MSDT[k][i] >0 then
18:                    ACWT[k]++;
19:                end if
20:            end for
21:        end for
22:    end for
23: end procedure
```

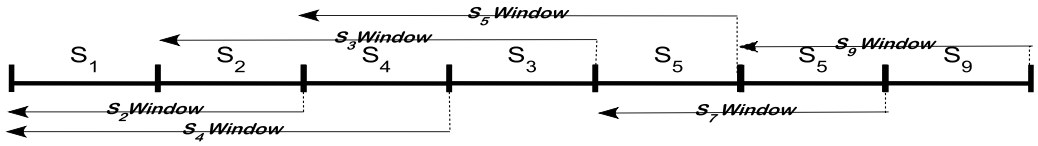
After that, we proceed through an EventStream partition (Definition 3) that builds a set of partially overlapping windows over the EventStreams using the CW table. Definition 3 specifies that each window shares the set of its elements with the window which precedes it except the last event which contains the reference service of the window.

Definition 3 (Partition) *A **Partition** builds a set of partially overlapping windows over an* EventStream*.*
Partition: CSLog → *(*Window*)**
S: EventStream*(bStream, eStream, sLog, CSocc)* → $\{w_i$*:*Window*; 0≤i<n\}:*

- w_1*.bWin = bStream and* w_n*.eWin = eStream,*
- $\forall w : window \in partition$*, e:*Event*= the last event in* w*,* width*(*w*)= ACWT[e.ServiceID],*
- *∀ 0≤i<n;* w_{i+1}*.wLog -* $\{$*the last event e:*Event *in* w_{i+1}*.wLog*$\} \subset w_i$*.wLog* $\wedge$ w_{i+1}*.wLog* $\neq w_i$*.wLog.*

In Figure 4 we have applied a partition over the EventStream of the running example presented in the EventStream 4 in Table 1. For example, the CW size of S_4 is equal to 4 because this service is in concurrence with two other services S_2 and S_3. And, the size of the CW of S_5 is equal to 3 because this service has two concurrent services S_3 and S_4 which precede it. We note that for each service in this EventStream its CW enables it to cover only its all causal preceding services.

Figure 4 EventStream partition



Finally, Algorithm 4 computes the final SDT. For each *concurrent window*, it computes for its reference (last) service the frequencies of its preceding services. The final SDT will be found by dividing each row entry by the frequency of the row's service.

Algorithm 4 Calculating final SDT

procedure FINALSDT($Wlog, \#, MSDT$)
input: Wlog: CSLog, #: Service Frequencies Table; $MSDT$: Marked Statistical Dependencies Table;
output: $FSDT$:Final Statistical Dependencies Table
Variable: $S_{reference}$: int; $S_{preceding}$: int; fWin: window; depFreq: int[][]; freq: int;

    $MSDT_{size} \leftarrow$ *Size-tab*($MSDT$); ▷ returns MSDT size
    for *all* win:window *in* partition(Wlog) **do**
        $S_{reference}$ = *last-service*(win); ▷ returns the last service's event
        fwin = *preceding-events*(win); ▷ returns "win" without the last event
        for *all* e:event *in* fwin.wLog **do**
            $S_{preceding}$= e.serviceId;
            if MSDT[$S_{reference}$, $S_{preceding}$]>0 **then**
                depFreq[$S_{reference}$, $S_{preceding}$]++;
            end if
        end for
    end for
    for int t_{ref}=0; $t_{ref} < MSDT_{size}$; t_{ref}++; **do**
        for int t_{pr}=0; $t_{pr} < MSDT_{size}$; t_{pr}++ **do**
            $FSDT$[t_{ref}, t_{pr}]= depFreq[t_{ref}, t_{pr}]/#t_{ref};
        end for
    end for
end procedure

Now by applying these algorithms, we can compute the final SDT (FSDT) which will be used in the next section to discover the patterns (Table 3). Note that, our approach adjust **dynamically**, through the width of CW, the process calculating services dependencies. Indeed, this width is sensible to concurrent behavior: it increases in case of concurrence and is "neutral" (equal to 2) in case of concurrent behavior absence. Thus, our algorithm adapts its behavior to the "concurrent" context. This strategy allows the improvement of the algorithm's complexity and runtime execution comparing to an analog patterns discovery [10] which uses an invariable concurrent window width. Indeed, the use of an invariable width could apply a width superior to 2 for nonconcurrent services or simply a non optimal width and then involve unnecessary computations increasing simply the algorithm's complexity.

Table 3 Final Statistical Dependencies Table (FSDT)

$P(x/y)$	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S_1	0	0	0	0	0	0	0	0	0
S_2	**1**	0	0	-1	0	0	0	0	0
S_3	0	**1**	0	-1	0	0	0	0	0
S_4	**1**	-1	-1	0	0	0	0	0	0
S_5	0	0	**1**	**1**	0	0	0	0	0
S_6	0	0	0	0	1	0	0	0	0
S_7	0	0	0	0	1	0	0	0	0
S_8	0	0	0	0	0	0	0	0	0
S_9	0	0	0	0	0	0.38	0.62	0	0

$$\#S_1 = \#S_2 = \#S_3 = \#S_4 = \#S_5 = \#S_9 = 100,$$
$$\#S_6 = 38, \#S_7 = 62, \#S_8 = 0$$

3.2 Control flow statistical properties

We have identified three kinds of behavior: sequential exclusive, conditional, and concurrent, which specify the patterns that we aim to discover. We have described these behavioral features by statistical properties using SDT (see Figure 3). We use theses properties to identify separately patterns from CS log. These properties bind a correlation link between log statistics represented in the SDT and patterns' main behavior using a set of corollaries deduced from Theorem 1.

We begin with the statistical exclusive dependency property (Corollary 1) which characterises a single sequential flow. The behavior of the *mutual* exclusive dependency between two services specifies that the execution of one of the two services depends only on the end of execution of the other, and the end of execution of the first service starts only the execution of the second.

Corollary 1 (*P1*: Mutual exclusive dependency property)
Let S_i and S_j two services. S_i and S_j describe a mutual *exclusive dependency property (*P1*) from S_i to S_j* **iff** *in terms of:*
- *services frequencies:* $\#S_i = \#S_j$
- *services dependencies :* $P(S_i/S_j) = 1 \wedge \forall(0 \leq k, l < n; k \neq j; l \neq i; P(S_i/S_k) = 0 \wedge P(S_l/S_j) = 0)$.

The parallel behavior (Corollary 2) inherits from a non causal relation. It specifies how, in terms of concurrence, the execution of a set of services. This set of services is located after a "fork" or before a "join" operator. We distinguished three types of parallel behavior:

1. **P2.1: Global concurrency** where in the same instantiation the services are performed simultaneously
2. **P2.2: Partial concurrency** where in the same instantiation we have at least a partial concurrent execution between the services
3. **P2.3: No concurrency** where there is no concurrency between the services

The conditional behavior (Corollary 3) specifies how the activation choice is carried out among a group of services after a "fork" operator or before a "join" operator. It defines a causal relation between a service and a group of services forming the operands of the "fork" operator or the "join" operator. We distinguished three types of conditional behavior:

Corollary 2 (*P2*: Concurrency property)
Let $\{S_i, 0 \leq i < n\}$ *a set of services that describes a:*

1. ***P2.1: Global concurrency*** *property* **iff** $\forall i, j; 0 \leq i < j < n; \#S_i = \#S_j \wedge P(S_i/S_j) = -1$
2. ***P2.2: Partial concurrency*** *property* **iff** $\exists i, j; 0 \leq i < j < n; P(S_i/S_j) = -1$
3. ***P2.3: No concurrency*** *property* **iff** $\forall i, j; 0 \leq i < j < n; \wedge P(S_i/S_j) \neq -1$

- **P3.1: Free choice** where a part of the group of services is executed according to the constraints and parameters of each instantiation.
- **P3.2: Single choice** where only one service is executed. The choice of this service depends on the constraints and parameters on the executed instance;
- **P3.3: No choice** where all services are executed for each instantiation.

Corollary 3 (*P3*: Choice property) *Let* A *a service and* $\{S_i, 0 \leq i < n\}$ *a group of services forming the operands of a "*fork*" operator or a "*join*" operator. A and* $\{S_i, 0 \leq i < n\}$ *describe a:*

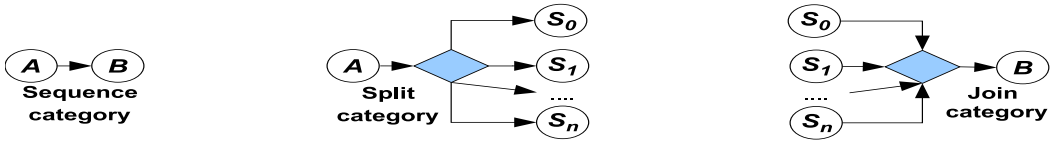
- ***P3.1: Free choice*** *property* **iff** *in terms of services frequencies we have* $(\#A \leq \Sigma_{i=0}^{n-1}(\#S_i)) \wedge (\#S_i \leq \#A)$ *and in terms of services dependencies we have:*
  - *In "*fork*" operator :* $\forall 0 \leq i < n; P(S_i/A) = 1$ *(S_i occurs certainly after A occurrence)*
  - *In "*join*" operator :* $1 < \Sigma_{i=0}^{n-1} P(A/S_i) < n$ *(A occurs certainly after some S_i occurrences "$1 <$", but not always after all S_i "$< n$")*
- ***P3.2: Single choice*** *property* **iff** *in terms of services frequencies we have* $(\#A = \Sigma_{i=0}^{n-1}(\#S_i)))$ *and in terms of services dependencies we have:*
  - *In "*fork*" operator :* $\forall 0 \leq i < n; P(S_i/A) = 1$ *(S_i occurs certainly after A occurrence)*
  - *In "*join*" operator :* $\Sigma_{i=0}^{n-1} P(A/S_i) = 1$ *(A occurs certainly after only one of S_i occurrences)*
- ***P3.3: No choice*** *property* **iff** *in terms of services frequencies we have* $\forall 0 \leq i < n$, $\#A = \#S_i$ *and in terms of services dependencies we have:*
  - *In "*fork*" operator :* $\forall 0 \leq i < n; P(S_i/A) = 1$ *(S_i occurs certainly after A occurrence)*
  - *In "*join*" operator :* $\forall 0 \leq i < n; P(A/S_i) = 1$ *(A occurs certainly after all S_i occurrences)*

3.3 Patterns discovering rules

Using the previous statistical properties, the last step is the patterns discovery through a set of rules. our approach provides a dynamic algorithm that builds iteratively global solution (i.e. global CS) based on local solutions (i.e. CS patterns). Each pattern has its own statistical rules which abstract statistically its causal and non-causal dependencies, and identify it in an unique manner. Our control flow mining rules are characterised by a "local" patterns discovery. Indeed, these rules proceed through a **local log analysing** that allows to **recover partial results** of mining patterns. To discover a particular pattern we need only events relating to pattern's elements. Thus, even using only fractions of CS log, we can discover correctly corresponding patterns (which their events belong to these fractions).

We divided the CSs patterns in three categories (c.f Figure 5) : sequence, split and join patterns. In the following, we present rules to discover the most useful patterns belonging to these three categories.

Figure 5 Patterns categories



3.3.1 Discovering sequence pattern In this category, we find only the sequence pattern (Table 4). In this pattern, the enactment of B depends only on the completion of service A. So we have used the statistical exclusive dependency property (Corollary 1) to ensure this relation linking B to A.

Table 4 Rules of sequence pattern

sequence	**Rules**
	$(\#B = \#A) \wedge$ $(P(B/A) = 1) \wedge \forall S_{0 \leq i < n} \neq A; P(B/S_i) \leq 0 \wedge \forall S_{0 \leq j < n} \neq B; P(S_j/A) \leq 0$

For instance, by applying the rules on this pattern over Table 3, we discover a sequence pattern linking S_2 and S_3. Indeed, $(\#S_2 = \#S_3)$ and $(P(S_2/S_3) = 1)$ and $\forall S_{0 \leq i < n} \neq S_2; P(S_3/S_i) \leq 0$ and $\forall S_{0 \leq j < n} \neq S_2; P(S_j/S_3) \leq 0$.

3.3.2 Discovering split *patterns* This category (Table 5) has a "fork" operator where a single thread of control splits into multiple threads of control which can be, according to the used pattern, executed or not. The dependency between services A and $\{S_i; 0 \leq i \leq n\}$ before and after "fork" operator differs in the three patterns of this category: xor-split, and-split, and or-split. These dependencies are characterised by the statistical choice properties (corollary 3). The xor-split pattern, where one of flows after "fork" operator is chosen, adopts the single choice property (P3.2). While and-split and xor-split patterns differentiate themselves through the no choice (P3.3) and free choice (P3.1) properties. Effectively, only a part of services are executed in or-split pattern after "fork" operator, while all S_i are executed in and-split pattern. The non-parallelism between S_i, in xor-split pattern are ensured by the no concurrency property P2.3 while the partial and the global parallelism in or-split and and-split patterns is identified through the application of the statistical partial and global concurrency properties P2.1 and P2.2. For instance, Table 3 indicates that we have an and-split pattern linking S_1, S_2 and S_4. In fact, there is a global parallelism between S_2 and S_4 $((P(S_4/S_2) = -1 \wedge P(S_2/S_4) = -1))$ and these services depend exclusively on S_1 $((P(S_4/S_1) = 1 \wedge P(S_2/S_1) = 1))$.

3.3.3 Discovering join *patterns* This category (Table 5) has a "join" operator where multiple threads of control merge in a single thread of control. The number of necessary branches for the activation of service B after "join" operator depends on the used pattern. To identify the three patterns of this category: xor-join, and-join and M-out-of-N-Join, we have analysed dependencies between S_i and B before and after "join" operator. Thus, the single choice (P3.2) and the no concurrency (P2.3) properties are used to identify xor-join pattern where two or more alternative branches come together without synchronisation and none of the alternative branches is ever executed in parallel. As for and-join pattern where multiple parallel services converge into one single thread of control, the no choice

Table 5 Rules of split and join patterns

split	**Rules**	join	**Rules**
(xor)	$(\Sigma_{i=0}^{n-1}(\#S_i)$=#A) $\wedge$ $(\forall 0 \leq i < n; P(S_i/A) = 1) \wedge$ $(\forall 0 \leq i \neq j < n; P(S_i/S_j) = 0)$	(xor)	$(\Sigma_{i=0}^{n-1}(\#S_i)$=#B) $\wedge$ $\Sigma_{i=0}^{n-1}$ P(B/S_i)=1) $\wedge$ $\forall 0 \leq i \neq j < n; P(S_i/S_j) = 0$
(and)	$((\forall 0 \leq i < n; \#S_i$=#A) $\wedge$ $(\forall 0 \leq i < n; P(S_i/A) = 1) \wedge$ $(\forall 0 \leq i \neq j < n\ P(S_i/S_j) = -1)$	(and)	$(\forall 0 \leq i < n; \#S_i$=#B) $\wedge$ $(\forall 0 \leq i < n; P(B/S_i) = 1) \wedge$ $(\forall 0 \leq i \neq j < n\ P(S_i/S_j) = -1)$
(or)	$(\#A \leq \Sigma_{i=0}^{n-1}(\#S_i)) \wedge$ $(\forall 0 \leq i < n; \#S_i \leq \#A)$ $(\forall 0 \leq i < n; P(S_i/A) = 1) \wedge$ $(\exists 0 \leq i \neq j < n; P(S_i/S_j) = -1)$	(M-out -of-N)	$(m * \#B \leq \Sigma_{i=0}^{n-1}(\#S_i))$ $\wedge (\forall 0 \leq i < n; \#S_i \leq \#B)$ $(m \leq \Sigma_{i=0}^{n-1} P(B/S_i) \leq n)$ $\wedge (\exists 0 \leq i \neq j < n; P(S_i/S_j) = -1)$

(P3.3) and the global concurrency (P2.3) are both used to discover this pattern. In contrary of the M-out-of-N-Join pattern, where we need only the termination of M services from the incoming n parallel paths to enact B, the concurrency between S_i is partial (P2.2) and the choice is free (P3.1). For instance, using FSDT table we mine a xor-join pattern linking S_6, S_7 and S_9. In fact, FSDT's entries of these services indicate a non concurrent behavior between S_6 and S_7 ($P(S_6/S_7)$=$P(S_6/S_7)$ $\neq$ -1) and the execution of S_9 depends on the termination of S_6 or S_7 ($P(S_9/S_6)$+$P(S_9/S_7)$= 1).

3.4 Coherent composition of the discovered patterns

The construction of the CS complete graph is made by linking one by one the discovered patterns. Indeed, in our approach we define the control flow as an union of patterns. We use rewriting rules (illustrated in Table 6) to bind the discovered patterns (terminals). We consider a composite service as word that has patterns as terminals (laterals). These terminals can be associative and commutative in the word constituting the composite service when applying the rewriting rules.

Discovering a pattern-oriented model ensures a sound and well-formed mined CS model. Therefore, by using this kind of model we are sure that the discovered CS model do not contain any deadlocks or other flow anomalies. Indeed, this set of rules allows to discover a coherent and well-formed pattern-oriented CS model. Consequently, by using these rewriting rules we are sure that the discovered patterns does not contain any incoherent flow. In fact in order to not have non-sense unions (disjoined patterns) or incoherent flows (deadlocks, liveness, etc.), the grammar of this rewriting rules system defines a language of coherent unions that reduces the discovered patterns to the final word $\mathcal{W}orkflow$. Concretely, RR1 to RR7 rules rewrite the discovered in federated expressions that are reduced thereafter in rules RR8 to RR19 in the final word $\mathcal{W}orkflow$. Thus, a discovered control flow is coherent ***iff*** the union of the corresponding discovered patterns is a word generated by this grammar. Concretely this grammar, which was specified for the set of the seven studied patterns, postulates that:

- A control flow should start with one of these patterns: sequence, and-split, or-split or xor-split (rewriting rules RR8, RR13, RR14, RR15, RR16, RR17, RR18, RR19).
- All patterns can be followed or preceding by the sequence pattern (rewriting rules RR8, RR9, RR10, RR11, RR12).
- An and-split pattern should be followed by one of these patterns: and-join, M-out-of-N, xor-join or sequence (rewriting rules RR10, RR13, RR14, RR15).

Table 6 Rewriting rules defining a coherent pattern composition grammar

RR1: *sequence*(a,b),$\{p_i\} \longrightarrow \mathcal{A}(a,b)$,$\{p_i\}$
RR2: *and-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{B}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR3: *or-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{C}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR4: *xor-split*$(a,b_1,b_2,...,b_n)$,$\{p_i\} \longrightarrow \mathcal{D}(a,b_1,b_2,...,b_n)$,$\{p_i\}$
RR5: *and-join*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{E}(a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR6: *M-out-of-N*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{F}\{a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR7: *xor-join*$(a_1,a_2,...,a_n,b)$,$\{p_i\} \longrightarrow \mathcal{G}(a_1,a_2,...,a_n,b)$,$\{p_i\}$
RR8: $\mathcal{A}(a,b)$, $\mathcal{A}(b,c)$,$\{p_i\} \longrightarrow \mathcal{A}(a,c)$,$\{p_i\}$
RR9: $\mathcal{A}(x,a)$, $\mathcal{F}sn(a,b_1,b_2,...,b_n)$, $\{p_i\} \longrightarrow \mathcal{F}sn(x,b_1,b_2,...,b_n)$,$\{p_i\}$
RR10: $\mathcal{F}sn(a,b_1,...,b_n)$, $\mathcal{A}(b_i,x)$, $\{p_i\} \longrightarrow \mathcal{F}sn(a_0,b_1,...,b_{i-1},x,b_{i+1},...,b_n)$,$\{p_i\}$
RR11: $\mathcal{A}(x,a_i)$, $\mathcal{J}nt(a_1,...,a_n,b)$, $\{p_i\} \longrightarrow \mathcal{J}nt(a_1,...,a_{i-1},x,a_{i+1},...,a_n,b)$,$\{p_i\}$
RR12: $\mathcal{J}nt(a_1,a_2,...,a_n,b)$, $\mathcal{A}(b,x)$, $\{p_i\} \longrightarrow \mathcal{J}nt(a_1,a_2,...,a_n,x)\{p_i\}$
RR13: $\mathcal{B}(a,b_1,b_2,...,b_n)$, $\mathcal{E}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR14: $\mathcal{B}(a,b_1,b_2,...,b_n)$, $\mathcal{F}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR15: $\mathcal{B}(a,b_1,b_2,...,b_n)$, $\mathcal{G}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR16: $\mathcal{C}(a,b_1,b_2,...,b_n)$, $\mathcal{F}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR17: $\mathcal{C}(a,b_1,b_2,...,b_n)$, $\mathcal{G}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR18: $\mathcal{D}(a,b_1,b_2,...,b_n)$, $\mathcal{G}(b_1,b_2,...,b_n,c)$, $\{p_i\} \longrightarrow \mathcal{A}(a,c)$, $\{p_i\}$
RR19: $\mathcal{A}(a,b)$, $\varepsilon \longrightarrow \mathcal{W}orkflow$

$\{p_i\}$ remaining terminal set
$\mathcal{F}sn = \mathcal{B} \vee \mathcal{C} \vee \mathcal{D}$
$\mathcal{J}nt = \mathcal{E} \vee \mathcal{F} \vee \mathcal{G}$

– An or-split pattern should be followed by one of these patterns M-out-of-N, xor-join or sequence (rewriting rules RR10, RR16, RR17).
– A xor-split pattern should only be followed by xor-join pattern or sequence pattern (rewriting rules RR10, RR18).

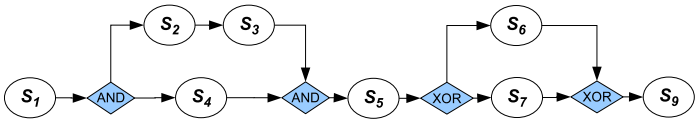
By applying the discovering rules (Tables 4 and 5) over the final SDT (Table 3) we discovered the composite service illustrated in Figure 6. We built the control flow as a patterns composition over this pattern word:

and-split(S_1,S_2,S_4), sequence(S_2, S_3), and-join(S_3,S_4,S_5), xor-split(S_5, S_6, S_7), xor-join(S_6, S_7, S_9).

Concretely by applying the rewriting rules (Table 6) to this word, we can combine these discovered patterns, by binding them in a coherent structure to rebuild and analyze the coherence of our discovered composite service:

and-split(S_1,S_2,S_4), sequence(S_2, S_3), and-join(S_3,S_4,S_5), xor-split(S_5, S_6, S_7), xor-join(S_6, S_7, S_9). $\longrightarrow_{RR1,RR2,RR3,RR4,RR5,RR6,RR7}$ $\mathcal{B}(S_1,S_2,S_4)$, $\mathcal{D}(S_5, S_6, S_7)$, $\mathcal{G}(S_6, S_7, S_9)$, $\mathcal{A}(S_2, S_3)$, $\mathcal{E}(S_3,S_4,S_5) \longrightarrow_{RR11}$ $\mathcal{B}(S_1,S_2,S_4)$, $\mathcal{D}(S_5, S_6, S_7)$, $\mathcal{G}(S_6, S_7, S_9)$, $\mathcal{E}(S_2,S_4,S_5) \longrightarrow_{RR18}$ $\mathcal{B}(S_1,S_2,S_4)$, $\mathcal{A}(S_5, S_9)$, $\mathcal{E}(S_2,S_4,S_5) \longrightarrow_{RR13}$ $\mathcal{A}(S_1,S_5)$, $\mathcal{A}(S_5, S_9) \longrightarrow_{RR8}$ $\mathcal{A}(S_1,S_9) \longrightarrow_{RR19}$ $\mathcal{W}orkflow$

Figure 6 CS discovered example



4 Composite service validation and re-engineering

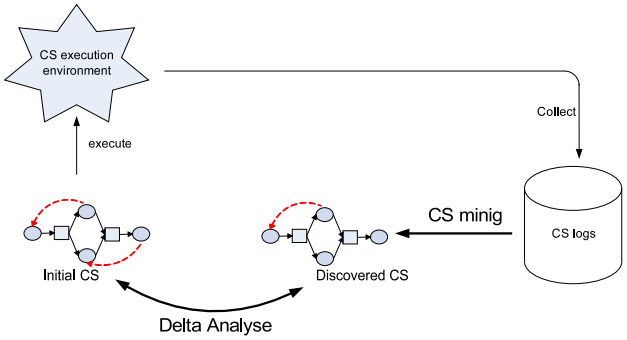
There exist two important process validation questions [11] : (1) *"Does our model reflect what we actually do ?"* and (2) *"Do we follow our model" ?*. Within the context of a business processes re-engineering, we address the question (2) to propose a set of improvement and correction tools based on the CS discovery results. We are interested, in particular, on the correction and the improvement of the CS's control flow. The goal is to provide an assistance tool to correct the CS design by applying semantic (adding services, suppressing services, and/or modifying pattern type) corrections. Our aim is to be close to the business and conceptual choices of CS designers and the evolution needs of CS users expressed during runtime and reported by the CS discovery results.

We will use the process discovery for a delta analysis (*c.f.* section 4.1), i.e. to compare the real operational process, represented by the discovered CS, with the initially designed CS model (for example, an ad-hoc composite web service orchestration). By comparing the initial CS with the discovered CS, the discrepancies between the two models can be detected and used to improve, in particular, the control flow. With this intention, we propose, thereafter, a set of actions which allow to correct or to remove, if necessary, any erroneous or useless flow and optimize the process execution (*c.f.* section 4.2). By erroneous or useless behavior, we mean any initially designed flow which is not necessary or which does not coincide with the execution reality expressing new users' needs or conceptual choices errors made in the first design phase. Indeed, this behavior can be simply expensive but also source of errors.

4.1 Delta Analysis

Concretely, we can use the CS discovery for Delta analysis, i.e, compare the "real" operational process, represented by the discovered CS, with the initially designed CS (for example, an ad-hoc composite web service orchestration). By comparing the initial CS with the discovered CS, we aim to detect discrepancies between the two models. Indeed, at run time users can deviate from the initially designed CS. Delta Analysis (Figure 7) between the initially designed and the discovered CS allow to monitor these deviations.

Figure 7 Delta analyse for CS re-engineering



Delta analysis uses comparison techniques to compare between the two models. Although the comparison techniques of process models do not constitute the core of our work but rather a tool,

we present, in the following, an overview of the existing solutions. Judging by the great number of equivalence concepts [12] this task is far from being insignificant. Most of business process comparison approach propose a node mapping technique rather than behavior mapping technique, i.e. the interest goes on the syntactic differences rather than on the semantic differences.

However, from a theoretical point of view, there are at least two approaches which also include a behavioral comparison. The first approach defined in [13, 14] uses the "behavior legacy". The second one is based on the process "changing regions". Based on the "behavior legacy" concept, Van der Aalst et al. developed the concept of the largest common divider and the smaller common multiple of two processes [14] as comparison tool. While the "changing regions" processing [15, 16] is obtained by comparing the two processes models and by extending the areas which were changed directly by the parts of the processes which are also affected by the change coming from the other process, i.e, the syntactically affected parts are extended with the semantically affected parts to produce the "changing regions".

4.2 Improving and correcting the control flow

Independently of the chosen comparison technique, a delta analysis process aims to detect the discrepancies between the discovered and the initial models. These discrepancies express the possible process model deviations. The analysis of these deviations is fundamental for a new re-engineering phase. Indeed, deviations can be exploited: (option 1) to motivate the process users to be closer to the initially designed process if the discrepancies do not express a real evolution, or (option 2) to correct and improve the process model to be as close as possible to the "execution" reality. In fact, some of these deviations become a current practice rather than to be a rare exception. In the following, we propose a set of tools to correct and improve the control flow according to option 2.

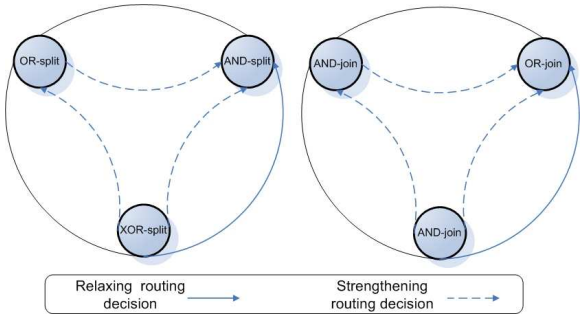
Thus, the discrepancies detected thanks to the delta analysis are used to improve, correct or remove, if necessary, all "erroneous or useless design". By "erroneous or useless design", we mean an initially designed flow which is not necessary or that does not coincide with the reality of execution and express evolution needs from the users or conceptual choices errors made at the initial design phases. In this case, the accuracy and the reliability of the initially designed process are uncertain and a re-engineering phase based on the discrepancies between the two models is required.

The correction and improvement actions related to the re-engineering phase depend on the discovered discrepancies. These actions should respect designers' and users' business needs. We distinguish between two kinds of discrepancies related to the flows and operators nature which allow:

- to suppress erroneous flows containing useless services. These useless services are not reported in logs and their related rows and columns are empty in SDT. Thereafter the discovered CS does not contain these services. Keeping these services in the CS can be merely expensive but also source of errors. For instance, by comparing the initial CS (Figure 1) with the discovered CS (Figure 6), S_8 does not exist any more in the discovered CS. This indicates that the payment by cash is never executed and can be removed from the initial CS model because it does not represent a "used" payment choice (or alternative). Indeed, the flow containing this service is not necessary for an optimal CS processing and does not coincide with the reality of execution and its maintenance can cause an additional cost, hence the proposal to remove it.
- to correct or improve operators nature expressing routing decisions. Indeed, you can detect discrepancies between the operator (xor, and, or) in the discovered and initially designed CSs. These discrepancies can express either a relaxation or a strengthener of the parameters related to the routing decisions specifying the choice performed over the set of services after the operator in the split patterns or before the operator in the join patterns (see Figure 8). The routing

decision relaxation (for instance, from and-join to or-join) expresses that the executed services will be wider after the operator in the split patterns and limited before the operator the join patterns than in the initially designed CS model. This relaxation could be a result of decisions' constraints dropping due new user's needs that imply to remove these constraints in a flexible and dynamic process execution. Respectively, the routing decision strengthening (for instance, from or-split to xor-split) expresses that the executed services will be limited after the operator in the split patterns and wider before the operator in the join patterns than in the initial designed model. This relaxation could be a result of adding decisions' constraints due new user's needs or specific execution environment's features that imply to add new constraints or to discover new features. For instance, we discover a xor-split pattern linking S_5, S_6 and S_7 (Figure 6), instead of an or-split in the initially designed CS (Figure 1). This discrepancy indicates that the users do not combine the payment by check and the payment by credit card and use exclusively one of them. This restriction of choice is induced by specific user's evolutions needs expressed through the CS execution and captured by the CS logs that we use to discover the "real" behavior. The transformation of the or-split pattern in the initially designed CS to the xor-split pattern yields a more efficient and accurate CS that sticks on users' behavior and avoids to implement unnecessary payment means (the combination of the payment by credit card and check) that can imply additional management costs.

Figure 8 Operators evolution



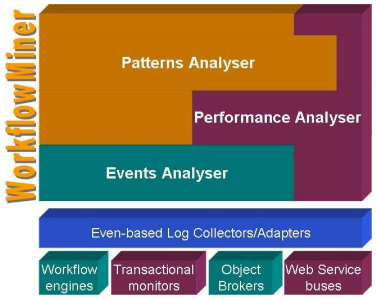
5 Implementation

Our approach, thought initially to discover process structure from their linear event streams, have been implemented within our prototype WorkflowMiner[8]. Since the single assumption of our approach is *the availability of execution logs* containing minimalist information (processes identifiers, process instances identifiers, services identifies, and time stamp for service instance event termination), application to web service reengineering was only conditioned with the collection of web service execution logs.

Figure 9 shows the general architecture of WorkflowMiner upon four main components: (1) *Event-based Log Collectors/Adapters*, (2) *Events Analyser*, (3) *Patterns Analyser*, and (3) *Performance Analyser*. WorkflowMiner components spirit inherits from 1^{st} order logic predicates based

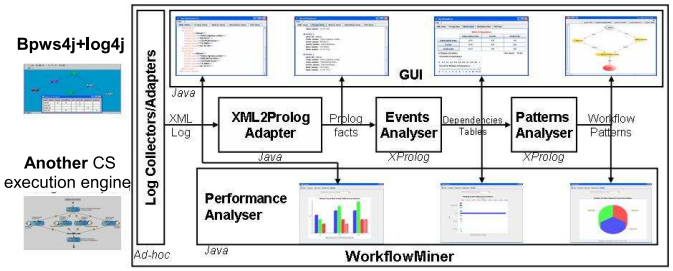
[8] WorkflowMiner demonstration can be downloaded at http://workflowminer.drivehq.com/workflowminer.avi

Figure 9 WorkflowMiner applicative and technical Architecture



reasoning, multidimensional database based business intelligence, and rich visual reporting. WorkflowMiner components are built on a panel of libraries and packages which the authors have either developed or integrated into WorkflowMiner. Data flow between WorkflowMiner components is described in the Figure 10. Starting from executions of composite web services, (1) event streams are gathered into an XML log. In order to be processed, (2) these log events are wrapped into a 1^{st} order logic format, compliant with definition 1. (3) Mining rules are applied on resulted 1^{st} order log events to discover structural patterns. We use a Prolog-based presentation for log events, and mining rules. (4) Discovered patterns are given to the web service designer so he/she will have a look on the analysis of his/her web services to restructure or redesign them either manually or semi-automatically.

Figure 10 WorkflowMiner Pipes and Filters Data Flow



Web service execution log Collectors/Adapters. Once intercepted and logged, web service interaction events (textual log lines, exchanged network messages), need to be adapted to be homogeneous, and usable by WorkflowMiner. Actually, event adapters translate, as an ETL (Event-Transform-Load), those non-structured Web service events into our WorkflowMiner compliant XML structures, and then into 1^{st} order logic Predicates Prolog form. WorkflowMiner event-based log collectors/adapters are developed using java xml parsers, ad-hoc adapters, and XProlog[9]. WorkflowMiner inputs are now ready to be processed within events, pattern, and performance analysers respecting the single assumption mentioned above.

Web service Events Analyser. Through statistical techniques developed in this paper, *final statistical dependencies* are inferred iteratively over event-based log. Thus, WorkflowMiner events anal-

[9] http://www.iro.umontreal.ca/∼vaucher/XProlog/

yser discovers basic web service interaction protocol based on intercepted web service interaction events. WorkflowMiner events analyser is developed using java xml parsers, and XProlog.

Web service Patterns Analyser. 1^{st} order logic predicates rules are used to discover a set of the most useful patterns which are divided into three categories: sequence pattern, split patterns (xor-split, and-split, or-split patterns) and join patterns (xor-join, and-join and M-out-of-N-Join patterns). Each structural pattern is expressed using statistical properties over the FSDT. Thus, WorkflowMiner patterns analyser discovers advanced web service interaction protocol that refines the basic web service interaction protocol. This advanced web service interaction protocol can be serialized to linear execution language (e.g. BPEL, BPML, etc.) replacing the ad-hoc composite web service orchestration. WorkflowMiner patterns analyser is developed using XProlog, and JGraph[10].

Web service Performance Analyser. WorkflowMiner *Performance Analyser* uses adapted event-based log, discovered causal dependencies, and discovered structural patterns to measure composite web service performance metrics (aka key performance indicators -KPI-). The theoretical, and experimental discussion of web service KPI is out of scope of this paper.

6 Discussion

Generally, previous formal approaches develop, using their modeling formalisms, a set of techniques to analyze the composition model and check its properties. [17] proposes a formal framework for modeling, specifying and analyzing the global behavior of Web services compositions. This approach models web services by mealy machines (finite state machines with input an output). Based on this formal framework, authors illustrate the unexpected nature of the interplay between local and global composite Web services. In [18], authors propose Petri net-based algebra for composing Web services. This formal model allows the verification of properties and the detection of inconsistencies both between and within services. Although powerful, the above formal approaches may fail, in some cases, to ensure optimum CS model even if they formally validate their composition models. This is because properties specified in the studied composition models may not coincide with the reality (i.e. effective CSs executions).

To the best of our knowledge, there are practically no approaches to web service compositions correction based on event-based logs, and in general there are very few contributions in this area. Prior art in this field is limited to estimating deadline expirations and exceptions prediction. [19,20] describe a tool set on top of HPs Process Manager which includes a so-called "BPI Process Mining Engine" and support business and IT users in managing process execution quality by providing several features, such as analysis, prediction, monitoring, control, and optimization. In [21], van der Aalst et al. check and quantify how much the actual behavior of a service, as recorded in message logs, conforms to the expected behavior as specified in a process model. Our approach differs from the above: using our patterns mining approach, we discover and prevent web service interactions anomalies. We start from CS logs and analyze them in order to re-engineer the CS model.

Obvious applications of process mining exist in model driven business process software engineering, both for bottom up approaches used in business process alignment [22,23], and for top down approaches used in workflow generation [24]. A number of research efforts in the area of workflow management have been directed for mining workflows models. This issue is close to that we propose in terms of discovery. The idea of applying process mining in the context of process management was first introduced in [25]. This work proposes methods for automatically deriving a formal model of a process from a log of events related to its executions and is based on workflow

[10] http://www.jgraph.com/

graphs. Cook and Wolf [26] investigated similar issues in the context of software engineering processes. They extended their work limited initially to sequential processes, to concurrent processes [27]. [28] presents an exhaustive survey of preceding process mining research works. Both areas of Process Mining and Business Process Reengineering are actively researched and considered as hot area in current business process management research activities. Process Mining covers different perspectives: the control flow perspective relates to the "How?" question, the organizational perspective to the "Who?" question, and the case perspective to the "What?" question. New issues in control flow perspective has been recently addressed by [29] that proposes genetic algorithms to tackle log noise problem or non-trivial constructs using a global search technique, and by [30] that uses Region based synthesis methods and compares their efficiency and usefulness. While the organizational perspective, [31] discovers information related to the social network in a process. And in the case perspective, [32] deals with the performance characteristics and business rules based on the case-related information about a Process.

In addition to WorkflowMiner, our approach has been implemented within the ProM framework [33], as a plug-in. ProM is a plug-in environment for process mining. The ProM framework is flexible with respect to the input and output format, and is also open enough to allow for the easy reuse of code during the implementation of new process mining ideas. This plug-in [34] helps us to provide detailed comparison [35] of our approach to other implemented mining tools. Our process mining approach can be distinguished by supporting **local discovery** through a set of control flow ming rules that are characterised by a "local" patterns discovery enabling **partial results** to be discovered. It recovers partial results from log fractions. Moreover, even if non-free choice (NFC) construct is mentioned as an example of a pattern that is difficult to mine, WorkflowMiner discovers M-out-of-N-Join pattern which can be seen as a generalisation of the useful Discriminator pattern that were proven to be inherently non free-choice. Recently, [36, 29] propose a complete solution that can deal with such constructs. Besides, our mining approach discovers **more complex features** with a better specification of "fork" operator (and-split, or-split, xor-split patterns) and "join" operator (and-join, M-out-of-N-Join, and M-out-of-N-Join patterns). We provided rules to discover the 7 most used patterns. But the adopted approach allows to enrich this set of patterns by specifying new statistical dependencies and their associated properties or by using the existing properties in new combinations. In an other side, our approach deals better with concurrency through the introduction of the "*concurrent window*" that proceeds **dynamically** with concurrence. Indeed, the size of the "*concurrent window*" is not static or fixed, but variable from one service to an other according to their concurrent behavior without increasing the computing complexity. It is trivial to establish that the algorithms describing our approach are of polynomial-complexity not exceeding the quadratic order $O(n^2)$. Indeed, the algorithms which we described do not contain recursive calls, and contain no more than 2 overlapping loops whose length is equal to the number of Events within an Eventstream.

In previous process mining works, the discovery of short loops and the handling of log noise are generally done in a separate log preprocessing step. We chose to not include it in this stage of our approach to not reduce our algorithm efficiency. We currently think in a more original method and less heavy approach to answer to these two points.

Our current work is about discovering complex patterns by using more metrics (*e.g.* entropy, periodicity, etc.) and by enriching the CS log. We are also interested in discovering more complex transactional characteristics of cooperative Composite service [37, 38]. In [39], we proposed a set of mining techniques to discover CS transactional flow in order to improve CS recovery mechanisms. Our work in [3] uses web services logs to enable the verification of behavioral properties in web service composition. The main focus has not been put on discovery but on verification. This means that given an event log and a formal property, we check whether the observed behavior matches the (un)expected/(un)desirable behavior. Recently, we have specified in [40] a combined approach

that describes a formal framework, based on Event Calculus to check the transactional behavior of CS before and after execution. Our approach provides a logical foundation to ensure transactional behavior consistency at design time and report recovery mechanisms deviations after runtime. In our future works, we hope to discover more complex patterns by using more metrics (e.g. entropy, periodicity, etc.) and by enriching the CS log structure.

References

1. Gombotz, R., Baïna, K., Dustdar, S.: Towards web services interaction mining architecture for e-commerce applications analysis. In: International Conference on E-Business and E-Learning (EBEL'05), Amman, Jordan (2005)
2. Fauvet, M.C., Dumas, M., Benatallah, B.: Collecting and querying distributed traces of composite service executions. In: On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, Springer-Verlag (2002) 373–390
3. Rouached, M., Gaaloul, W., van der Aalst, W.M.P., Bhiri, S., Godart, C.: Web service mining and verification of properties: An approach based on event calculus. In Meersman, R., Tari, Z., eds.: OTM Conferences (1). Volume 4275 of Lecture Notes in Computer Science., Springer (2006) 408–425
4. Punin, J., Krishnamoorthy, M., Zaki, M.: Web usage mining: Languages and algorithms. In: Studies in Classification, Data Analysis, and Knowledge Organization. Springer-Verlag (2001)
5. Baglioni, M., Ferrara, U., Romei, A., Ruggieri, S., Turini, F.: Use soap-based intermediaries to build chains of web service functionality (2002)
6. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9) (2004) 1128–1142
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addition-Wesley, Massachusetts (1994)
8. van der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (2003) 5–51
9. Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998)
10. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery **1**(3) (1997) 259–289
11. Cook, J.E., Wolf, A.L.: Software process validation: quantitatively measuring the correspondence of a process to a model. ACM Transactions on Software Engineering and Methodology (TOSEM) **8**(2) (1999) 147–176
12. v. Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3) (1996) 555–600
13. Basten, T., van der Aalst, W.M.P.: Inheritance of behavior. J. Log. Algebr. Program. **47**(2) (2001) 47–145
14. van der Aalst, W.M.P., Basten, T.: Identifying commonalities and differences in object life cycles using behavioral inheritance. In: ICATPN '01: Proceedings of the 22nd International Conference on Application and Theory of Petri Nets, London, UK, Springer-Verlag (2001) 32–52
15. van der Aalst, W.M.P.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers **3**(3) (2001) 297–317
16. Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: COOCS. (1995) 10–21
17. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: Proceedings of the twelfth international conference on World Wide Web, ACM Press (2003) 403–410
18. Hamadi, R., Benatallah, B.: A petri net-based model for web service composition. In: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003, Australian Computer Society, Inc. (2003) 191–200
19. Sayal, M., Casati, F., Shan, M., Dayal, U.: Business process cockpit. Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02) (2002) 880–883

20. Grigori, D., Casati, F., Castellanos, M., Dayal, U., Sayal, M., Shan, M.C.: Business process intelligence. Comput. Ind. **53**(3) (2004) 321–343
21. van der Aalst, W., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, H.: Conformance checking of service behavior. ACM Transactions on Internet Technology (TOIT), Special issue on Middleware for Service-Oriented Computing (2007)
22. van der Aalst, W.M.P.: Business alignment: Using process mining as a tool for delta analysis. In: CAiSE Workshops (2). (2004) 138–145
23. Benatallah, B., Casati, F., Toumani, F.: Analysis and management of web service protocols. In: ER. (2004) 524–541
24. Baïna, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: CAiSE. (2004) 290–306
25. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. Lecture Notes in Computer Science **1377** (1998) 469–498
26. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology (TOSEM) **7**(3) (1998) 215–249
27. Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 35–45
28. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow mining: a survey of issues and approaches. Data Knowl. Eng. **47**(2) (2003) 237–267
29. de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic process mining: an experimental evaluation. Data Min. Knowl. Discov. **14**(2) (2007) 245–304
30. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 375–383
31. van der Aalst, W.M.P., Reijers, H.A., Song, M.: Discovering social networks from event logs. Computer Supported Cooperative Work **14**(6) (2005) 549–593
32. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: An approach based on temporal logic. In Meersman, R., Tari, Z., Hacid, M.S., Mylopoulos, J., Pernici, B., Babaoglu, Ö., Jacobsen, H.A., Loyall, J.P., Kifer, M., Spaccapietra, S., eds.: OTM Conferences (1). Volume 3760 of Lecture Notes in Computer Science., Springer (2005) 130–147
33. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W.E., Weijters, A.J.M.M.: Prom 4.0: Comprehensive support for *eal* process analysis. In Kleijn, J., Yakovlev, A., eds.: ICATPN. Volume 4546 of Lecture Notes in Computer Science., Springer (2007) 484–494
34. Gaaloul, W., Godart, C.: A workflow mining tool based on logs statistical analysis. In Zhang, K., Spanoudakis, G., Visaggio, G., eds.: SEKE. (2006) 595–600
35. Baïna, K., Gaaloul, W., Khattabi, R.E., A.Mouhou: Workflowminer: a new workflow patterns and performance analysis tool. In: 18th International Conference on Advanced Information Systems Engineering (CAiSE'06) Forum, Luxembourg, Grand-Duchy of Luxembourg (2006)
36. Wen, L., van der Aalst, W.M.P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Min. Knowl. Discov. **15**(2) (2007) 145–180
37. Gaaloul, W., Bhiri, S., Godart, C.: Discovering workflow transactional behaviour event-based log. In: 12th International Conference on Cooperative Information Systems (CoopIS'04). LNCS, Larnaca, Cyprus, Springer-Verlag (2004)
38. Gaaloul, W., Godart, C.: Mining workflow recovery from event based logs. In: Business Process Management. Volume 3649. (2005) 169–185
39. Bhiri, S., Gaaloul, W., Godart, C.: Discovering and improving recovery mechanisms of compositeweb services. In: ICWS, IEEE Computer Society (2006) 99–110
40. Gaaloul, W., Hauswirth, M., Rouached, M., Godart, C.: Verifying composite service recovery mechanisms: A transactional approach based on event calculus. In: 15th International Conference on Cooperative Information Systems CoopIS07. (November, 2007)